**Zelle 4e Chapter 7 Coding Assignment**

**General Instructions**

My expectations for your work on coding assignment exercises will grow as we progress through the course. In addition to applying any new programming techniques that have been covered in the current chapter, I will be expecting you to follow all of the good programming practices that we have adopted in the preceding weeks. Here is a quick summary of good practices that we have covered so far:

- Include a Python Docstring that describes the intent of the program.
- Place your highest-level code in a function named *main*.
- Include a final line of code in the program that executes the *main* function.
- Follow all PEP-8 Python coding style guidelines enforced by the PyCharm Editor. For example, place two blank lines between the code making up a function and the code surrounding that function.
- Choose names for your variables that are properly descriptive.
- Define CONSTANT_VALUES and use them in place of *magic numbers*.
- Always use f-strings for string interpolation and number formatting.
- When processing items from Python lists and tuples, unpack the values into variables with meaningful variable names to avoid using indexed expressions in your code.
- Remember that your program should behave reasonably when it is not given any input. This might be the result of the user pressing enter at a console prompt.
- Model your solution after the code that I demonstrate in the tutorial videos.
- Make sure that your test input/output matches the sample provided.
- All functions that are not *main()* should have descriptive, action-oriented names.
- All functions should be of reasonable size.
- All functions should have high *cohesion*, and low *coupling*.
- Remember to test your program thoroughly before submitting your work.
- Your code must pass all relevant test cases. Make sure that it passes tests at the boundaries created by *if, else*, and *elif* conditions in your program (boundary value tests).
- Use of the *break* statement is allowed.
- Use of the *continue* statement is forbidden.

**This Assignment Has a Different Approach Toward Tutorial Videos**
There are no dedicated tutorial videos for this coding assignment. Instead, each exercise of this assignment has been associated with one of the sample programs that were included in the download named *Demonstrations to Supplement Zelle 4e Chapter 7*.

The instructions for each exercise below identify which sample program to consult. The instructions also provide a reference to a place in the Beyond the Textbook lecture videos (timecode provided) where that sample program is discussed. So, you may use that portion of the lecture video as your tutorial for the exercise.

**Exercise 1 (Regular)**
Create a program named *flexible_float_adder.py*. When creating this program, you may use the sample program *_05_flexible_integer_adding_machine.py* as a model. This sample program is discussed in the *Beyond the Textbook* lecture at 05:42.

Your program should be different in the following respects:

1. The user should be prompted to enter float values.

2. The program should accumulate float values.

3. When printing the accumulated sum, the program should display 2 decimal places.

You are only expected to test this program using manual unit testing.

When running a manual unit test where the user provides no input, you should expect the following input/output on your console:

```
Welcome to The Flexible Float Adding Machine.

Please enter a float value to be added (<Enter> to stop):

No entries were provided.
```

When running a manual unit test with more typical input, you should expect the following input/output on your console:

```
Welcome to The Flexible Float Adding Machine.

Please enter a float value to be added (<Enter> to stop): 1.11
Please enter a float value to be added (<Enter> to stop): 2.22
Please enter a float value to be added (<Enter> to stop): 4.44
Please enter a float value to be added (<Enter> to stop):

The sum of these 3 entries is 7.77
```

**Exercise 2 (Regular)**
Create a program named *usable_float_adder.py*. When creating this program, you may use the sample program *_08_usable_integer_adding_machine.py* as a model. This sample program is discussed in the *Beyond the Textbook* lecture at 07:08.

Your program should be different in the following respects:

1. The user should be prompted to enter float values.

2. The program should accumulate float values.

3. When printing the accumulated sum, the program should display 2 decimal places.

You are only expected to test this program using manual unit testing.

When running a manual unit test where the user provides no input, you should expect the following input/output on your console:

```
Welcome to The Usable Float Adding Machine.

Please enter a float value to be added (<Enter> to stop):

No entries were provided.
```

When running a manual unit test where the user provides bad input, you should expect the following input/output on your console:

```
Welcome to The Usable Float Adding Machine.

Please enter a float value to be added (<Enter> to stop): Hi, Mom!
A float value was expected. You entered Hi, Mom!
Please enter a float value to be added (<Enter> to stop): 1.11
Please enter a float value to be added (<Enter> to stop): 2.22
Please enter a float value to be added (<Enter> to stop): 4.44
Please enter a float value to be added (<Enter> to stop):

The sum of these 3 entries is 7.77
```

When running a manual unit test where the user provides typical input, you should expect the following input/output on your console:

```
Welcome to The Usable Float Adding Machine.

Please enter a float value to be added (<Enter> to stop): 1.11
Please enter a float value to be added (<Enter> to stop): 2.22
Please enter a float value to be added (<Enter> to stop): 4.44
Please enter a float value to be added (<Enter> to stop):

The sum of these 3 entries is 7.77
```

**Exercise 3 (Regular)**
Create a program named *rock_paper_scissors.py*. When creating this program, you may use the sample program *_20_playing_a_game_using_while.py* as a model. This sample program is discussed in the *Beyond the Textbook* lecture at 37:04.

Your program should be different in the following respects:

1. It will play *The Rock-Paper-Scissors Game* rather than *The Simple Coin Flipping Game.*

2. This program should implement the rules of the rock-paper-scissors game as presented in https://wrpsa.com/the-official-rules-of-rock-paper-scissors/ .

For this exercise, you are only expected to test your program using manual unit testing.

The following is a simple example of output that you should expect on your console when running a manual unit test:

```
Welcome to The Rock-Paper-Scissors Game.

Player A throws SCISSORS. Player B throws PAPER.
Player A wins.
```

The following is another simple example of output that you should expect on your console when running a manual unit test:

```
Welcome to The Rock-Paper-Scissors Game.

Player A throws SCISSORS. Player B throws ROCK.
Player B wins.
```

The following is a more complex example of output that you should expect on your console when running a manual unit test:

```
Welcome to The Rock-Paper-Scissors Game.

Player A throws ROCK. Player B throws ROCK.
Player A throws SCISSORS. Player B throws SCISSORS.
Player A throws ROCK. Player B throws ROCK.
Player A throws PAPER. Player B throws PAPER.
Player A throws SCISSORS. Player B throws PAPER.
Player A wins.
```

**Exercise 4 (Challenge)**
Using the techniques that we practiced in Chapter 6, create an automated unit test module for *rock_paper_scissors.py* named *test_rock_paper_scissors.py* using Pytest.

While automated unit tests of the dialog between the program and the user at the console are possible, they are beyond the beginner use of Pytest that we are taking in this course. So, that portion of the testing will remain manual.

Nevertheless, you should be able to create automated unit tests for any portion of your code that is not managing the user dialog and that is contained in a separate function.

Given that it is easier to create unit tests for code that has been factored into a function, you might want to consider refactoring your program to move more code into functions to get more automated test coverage.

**Tools**
Use PyCharm to create and test all Python programs.

**Submission Method**
Follow the process that I demonstrated in the tutorial video on submitting your work. This involves:

- Locating the properly named directory associated with your project in the file system.
- Compressing that directory into a single .ZIP file using a utility program.
- Submitting the properly named zip file to the submission activity for this assignment.

**File and Directory Naming**
Please name your Python program files as instructed in each exercise. Please use the following naming scheme for naming your PyCharm project:

```
surname_givenname_exercises_zelle_4e_chapter_07
```

If this were my own project, I would name my PyCharm project as follows:

```
trainor_kevin_exercises_zelle_4e_chapter_07
```

Use a zip utility to create one zip file that contain the PyCharm project directory. The zip file should be named according to the following scheme:

```
surname_givenname_exercises_zelle_4e_chapter_07.zip
```

If this were my own project, I would name the zip file as follows:

```
trainor_kevin_exercises_zelle_4e_chapter_07.zip
```

**Due By**
Please submit this assignment by the date and time shown in the Weekly Schedule.

**Last Revised**
2025-09-21